Constraint-Based Controller Synthesis in Non-Deterministic and Partially Observable Domains

Cédric Pralet and **Gérard Verfaillie** and **Michel Lemaître** and **Guillaume Infantes**[1]

Abstract. Controller synthesis consists in automatically building controllers taking as inputs observation data and returning outputs guaranteeing that the controlled system satisfies some desired properties. In system specification, these properties may be safety properties specifying that some conditions must always hold. In planning, they express that the evolution of the controlled system must terminate in a goal state. In this paper, we propose a generic approach able to synthesize memoryless or finite-memory controllers for both safety-oriented and goal-oriented control problems. This approach relaxes some restrictive assumptions made by existing work on controller synthesis with non-determinism and partial observability and is shown to induce potentially significant gains. The proposed "Simulate and Branch" algorithm consists in exploring the possible evolutions of the controlled system and in adding new control elements when uncovered states are discovered. The approach developed is constraint-based in the sense that control problems are formulated using the flexibility of constraint programming languages and that our implementation uses the Gecode constraint programming library.

1 INTRODUCTION

Controller synthesis consists in automatically building controllers that take as inputs observation data from a system to be controlled and return as outputs actions to be executed that guarantee that the controlled system satisfies some desired properties.

In system specification, these properties may be safety properties specifying that a condition must hold at any step over an infinite horizon. The associated synthesis problem can be hence referred to as *safety-oriented control*. More general properties expressed using temporal logic [4] are not considered in this paper.

In planning, the properties to be satisfied express that the controlled system must reach a goal state and halt in that state. The associated synthesis problem can be referred to as *goal-oriented control*. Related planning frameworks are contingent and conformant planning [2] for non-deterministic domains, and completely or partially observable Markov Decision Processes (MDP/POMDP) [9, 6] for stochastic domains.

When all attributes of the system to be controlled are observable, the controllers synthesized take the form of *memoryless controllers* mapping the last observation to an action. Otherwise, when some attributes remain unobserved, controllers take the form of *full-recording controllers*, mapping the belief state (the set of states consistent with all past observations) to an action [2, 1]. In goal-oriented contexts, these controllers can be transformed into *conditional plans* [1]. Intermediate approaches between memoryless and full-recording controllers were proposed for POMDP [7, 8] and recently for planning with non-determinism [3]. The principle is to synthesize *finite-memory controllers*, also referred to as finite-state controllers, which use a size-limited memory to record information related to the past. Compared to belief-state controllers, finite-state controllers are more reactive because they do not require any online maintenance of the belief state. Compared to full-recording controllers in general, they may be more compact.

In this paper, we propose a generic approach able to synthesize memoryless or finite-memory controllers for both safety-oriented and goal-oriented control problems in non-deterministic and partially observable domains. Compared to existing work in non-deterministic planning, we relax restrictive assumptions present in [3]: we do not assume that actions have no preconditions or that action effects are deterministic, and the controllers synthesized are guaranteed to terminate in a goal state, and not only to reach a goal state.

The search algorithm used, called *Simulate and Branch*, consists in exploring the possible evolutions of the system controlled (simulation phase) and in adding decisions when states uncovered by the policy are discovered (branching phase). The approach proposed is compared with [3] and with the MBP planner [1].

Last, the underlying models and algorithms defined are constraint-based. The idea is to use the expressiveness of a constraint programming framework to ease the modeling task, and the efficiency of associated algorithms to speed the search. The algorithms developed use the Gecode constraint programming library [5]. The techniques introduced are however not restricted to constraint-based models.

The paper is organized as follows: we first define control problems considered (Sections 2-3), then present the *Simulate and Branch* algorithm (Section 4), and last give experimental results (Section 5).

2 CONTROLLER SYNTHESIS FRAMEWORK

In this paper, the attributes of the controlled system are modeled by a finite set S of variables. The outputs of the controller are similarly modeled by a finite set C of variables. All variables $v \in S \cup C$ are assumed to have a finite domain of values denoted $\mathbf{d}(v)$. Given a set of variables V, $\mathbf{d}(V)$ denotes the Cartesian product of the domains of the variables in V (assuming an order on the variables in V so that this Cartesian product is well defined). Each element s (resp. c) in $\mathbf{d}(S)$ (resp. $\mathbf{d}(C)$) is called a *state* (resp. a *control*).

At a given step, the current state of the controlled system may be only partially known. We consider that the set S of state variables is partitioned into a set O of observable variables and a set $S \setminus O$ of unobservable variables. Each element o in $\mathbf{d}(O)$ is called an *observation*. Given a state s, $\mathbf{o}(s)$ denotes the assignment of the variables of O in s, that is the observation associated with s.

[1] ONERA, Toulouse, France, email: firstname.lastname@onera.fr

We consider a framework involving non-determinism both in the initial state and in the possible effects of the controller outputs. This non-determinism is modeled by two relations: an *initialization* relation I, which contains assignments s corresponding to possible initial states, and a *transition* relation T, which contains triples (s, c, s') such that s' is a possible successor of s when control c is performed.

Last, preconditions can be imposed on controller outputs. These preconditions may depend on both observable and unobservable attributes. They express that in a given situation, a controller output is physically impossible or forbidden by the modeler. Such preconditions are modeled using a *feasibility* relation F, which contains pairs (s, c) such that control c is feasible in state s.

In the models developed, relations I, T, and F are expressed as sets of constraints. In the following, given a relation $\mathcal{R}$ over a set of variables V, "$v \in \mathcal{R}$" is also denoted "$\mathcal{R}(v) = true$". We hence use $I(s) = true$, $T(s, c, s') = true$, and $F(s, c) = true$ to denote $s \in I$, $(s, c, s') \in T$, and $(s, c) \in F$ respectively.

The only assumption made on the previous elements is that a feasible control cannot block the evolution of the system: $\forall s \in \mathbf{d}(S)$, $\forall c \in \mathbf{d}(C)$, $F(s, c) \rightarrow (\exists s' \in \mathbf{d}(S), T(s, c, s'))$. Such an assumption is undemanding since, if it does not hold, it suffices to replace relation F by $\{(s, c) \mid F(s, c) \wedge (\exists s' \in \mathbf{d}(S), T(s, c, s'))\}$. All previous elements are gathered in the notion of *control model*.

Definition 1 *A* control model *is a tuple* (S, O, C, I, T, F) *such that:*

- S *is a finite set of finite-domain variables called* state variables*;*
- $O \subset S$ *is a set of* observable state variables*;*
- C *is a finite set of finite-domain variables called* control variables*;*
- $I \subset \mathbf{d}(S)$ *is the* initialization relation*;*
- $T \subset \mathbf{d}(S) \times \mathbf{d}(C) \times \mathbf{d}(S)$ *is the* transition relation*;*
- $F \subset \mathbf{d}(S) \times \mathbf{d}(C)$ *is the* feasibility relation*;*
- $\forall s \in \mathbf{d}(S)$, $\forall c \in \mathbf{d}(C)$, $F(s, c) \rightarrow (\exists s' \in \mathbf{d}(S), T(s, c, s'))$.

We then define a decision policy Π for a control model as a memoryless controller, mapping the last observation made $o \in \mathbf{d}(O)$ to a control $c \in \mathbf{d}(C)$. $\Pi(o) = c$ means that the output of the controller is c when observation o is made. Such policies are functional since they specify a unique possible output for each observation. A policy can be *partial* in the sense that $\Pi(o)$ can be undefined for some $o \in \mathbf{d}(O)$. Partial policies are useful to define the controller behavior only on the set of reachable states of the system.

Definition 2 *A* policy *for a control model* (S, O, C, I, T, F) *is a partial function* $\Pi : \mathbf{d}(O) \rightarrow \mathbf{d}(C)$. *The domain of a policy* Π *is defined as* $\mathbf{d}(\Pi) = \{o \in \mathbf{d}(O) \mid \Pi(o)$ *defined*$\}$.

Policies introduced in Definition 2 are memoryless since all past observations, but the current one, are not considered to determine a controller output. An opposite approach considers belief-state based policies $\Pi : 2^{\mathbf{d}(S)} \rightarrow \mathbf{d}(C)$ associating a controller output with a set of possible current states computed from all past observations. As shown in [3], an intermediate approach considers finite-state controllers. Such controllers maintain an internal state number $q \in [1..N]$, with N a fixed integer. They are defined by mappings $(o, q) \rightarrow (c, q')$ expressing that when the controller makes observation o and is in internal state q, it outputs control c and changes its internal state to q'. Thanks to internal memory $q \in [1..N]$, the controller may record features concerning past observations, and some problems which do not admit memoryless controllers admit finite-state controllers.

In order to model finite-state controllers with N internal states for a control model $M = (S, O, C, I, T, F)$, it actually suffices to consider a new control model $M' = (S', O', C', I', T', F)$ where $S' = S \cup \{qs\}$, $O' = O \cup \{qs\}$, $C' = C \cup \{qc\}$, $I' = I \wedge \{qs = 1\}$, and $T' = T \wedge \{qs' = qc\}$, and where qs and qc are new variables of domain $[1..N]$. The idea is to introduce a new observable state variable qs and a new control variable qc. Given the definitions of O' and C', decision policies in M' take the form $\Pi : \mathbf{d}(O) \times \mathbf{d}(qs) \rightarrow \mathbf{d}(C) \times \mathbf{d}(qc)$. As the value of qs is directly controlled thanks to transition constraint $qs' = qc$, it is then as if the controller contained mappings of the form $(o, qs) \rightarrow (c, qs')$. Initialization constraint $qs = 1$ sets arbitrarily the first internal state.

In the following, we consider that control models always contain one variables $qs \in O$ *and one variable* $qc \in C$*, both of domain* $[1..N]$. N is the maximum memory of the controller. It equals 1 for memoryless controllers (a unique possible internal state).

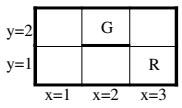

Figure 1. Robot control problem

Given a policy Π, it is possible to define the set of trajectories and the set of reachable states induced by Π. We are also interested in applicable policies which specify only feasible decisions. These elements are formalized below.

Definition 3 *Let* Π *be a policy for a control model* (S, O, C, I, T, F). *A* trajectory *induced by* Π *is a sequence* $[s_0, \ldots s_n]$ *such that (i)* $I(s_0) = true$*, (ii) for all* $i \in [1..n]$*,* $\mathbf{o}(s_{i-1}) \in \mathbf{d}(\Pi)$ *and* $T(s_{i-1}, \Pi(\mathbf{o}(s_{i-1})), s_i)$ *hold, and (iii)* $\mathbf{o}(s_n) \notin \mathbf{d}(\Pi)$ *if* $n < +\infty$*. When* $n < +\infty$ *(resp.* $n = +\infty$*), the trajectory is said to be finite (resp. infinite). The set of* reachable states *associated with* Π*, denoted* $A(\Pi)$*, is the set of states appearing in at least one trajectory induced by* Π.

Definition 4 *A policy* Π *for a control model* (S, O, C, I, T, F) *is said to be* applicable *if and only if for every reachable state* $s \in A(\Pi)$*,* $(\mathbf{o}(s) \in \mathbf{d}(\Pi)) \rightarrow F(s, \Pi(\mathbf{o}(s)))$.

To illustrate the controller synthesis framework, let us consider the example of a robot in a grid given in Figure 1. Initially, the robot is in one of the two positions of x-coordinate 2. At each step, it can observe the presence of walls around its current cell and move north, south, east, or west. To model this problem, we introduce the set of state variables $S = \{x, y, w_N, w_S, w_E, w_W, qs\}$ where x, y are variables of domains $\mathbf{d}(x) = [1..3]$ and $\mathbf{d}(y) = [1..2]$ representing the current xy-coordinates, w_N, w_S, w_E, w_W are boolean variables representing the presence of walls on the north, south, east, and west respectively, and qs is a variable of domain $\mathbf{d}(qs) = [1..N]$ representing the internal state of the controller. Position (x, y) is not directly observable, hence the set of observable variables is $O = \{w_N, w_S, w_E, w_W, qs\}$. The set of control variables is $C = \{m, qc\}$, where m has domain $\mathbf{d}(m) = \{m_N, m_S, m_E, m_W\}$ and represents the move performed at each step (north, south, east, or west), and qc has domain $\mathbf{d}(qc) = [1..N]$ and commands the evolution of the controller internal state. Note that even if all decisions have a deterministic effect in this example, we do not make such an assumption in the general framework.

The initialization relation (I), the relation transition (T), and the feasibility relation (F) are given by the following set of constraints, in which a primed variable represents the value of that variable at the next step (after the control):

$$I \quad \{\ x = 2, w_N, w_S, \neg w_E, \neg w_W, qs = 1$$

$$T \quad \begin{cases} x' = x + (m = m_E) - (m = m_W) \\ y' = y + (m = m_N) - (m = m_S) \\ w'_N \leftrightarrow (y' = 2 \vee (y' = 1 \wedge x' = 2)),\ w'_E \leftrightarrow (x' = 3) \\ w'_S \leftrightarrow (y' = 1 \vee (y' = 2 \wedge x' = 2)),\ w'_W \leftrightarrow (x' = 1) \\ qs' = qc \end{cases}$$

$$F \quad \begin{cases} w_N \rightarrow (m \neq m_N),\ w_S \rightarrow (m \neq m_S) \\ w_E \rightarrow (m \neq m_E),\ w_W \rightarrow (m \neq m_W) \end{cases}$$

An example of memoryless policy ($N = 1$) is Π_1 defined by:

$$\begin{array}{lrcl} \Pi_1 : & w_N, w_S, qs = 1 & \rightarrow & m = m_W, qc = 1 \\ & w_N, w_W, qs = 1 & \rightarrow & m = m_E, qc = 1 \\ & w_S, w_W, qs = 1 & \rightarrow & m = m_N, qc = 1 \end{array}$$

We omit negative literals. So, the first line corresponds to $w_N, w_S, \neg w_E, \neg w_W, qs = 1 \rightarrow m = m_W, qc = 1$. An example of finite-memory policy (with $N = 2$) is Π_2 defined by:

$$\begin{array}{lrcl} \Pi_2 : & w_N, w_S, qs = 1 & \rightarrow & m = m_E, qc = 1 \\ & w_N, w_E, qs = 1 & \rightarrow & m = m_W, qc = 2 \\ & w_S, w_E, qs = 1 & \rightarrow & m = m_N, qc = 1 \end{array}$$

3 CONTROL PROBLEMS

Given a control model, several requirements can be imposed on the possible evolutions of the controlled system. In this paper, three kinds of control problems are considered: *goal-oriented* control problems, *safety-oriented* control problems, and a combination of these two. Other problems could be considered, such as goal reachability in a bounded number of steps.

Goal-oriented control problems In such problems, the objective is to find an applicable policy so that all trajectories terminate in a goal state *i.e.*, reach a goal state and stop. The goal is defined by a *goal* relation G containing states s satisfying it. The distinction between "terminate" and "reach" matters when the goal relation holds on non-observable attributes.

Definition 5 *A* goal-oriented *control problem is a pair* (M, G) *with* M *a control model and* $G \subset \mathbf{d}(S)$ *the goal relation. A solution to this problem is an applicable policy* Π *for* M *such that all trajectories* $[s_0, \ldots, s_n]$ *induced by* Π *are finite and verify* $G(s_n) = true$.

For the robot example, let us assume that the goal is to terminate at position $(2, 2)$ (marked G on the figure). The goal-oriented control problem is then (M, G), with M the control model defined previously and G the goal relation defined by constraints $x = 2 \wedge y = 2$.

Policy Π_1, previously defined, is not a solution to this problem because, starting from initial state $sa : (x = 2, y = 2, qs = 1)$, it induces the infinite loopy trajectory $(sa, sb, sa, sb, \ldots)$ with $sb : (x = 1, y = 2, qs = 1)$. Intuitively, with Π_1, the controller never knows whether or not the goal state is reached. It can even be shown that no memoryless controller is solution to this problem, since for such a controller, positions $(2, 1)$ and $(2, 2)$ are always ambiguous.

Policy Π_2, which has memory $N = 2$, is a solution. Indeed, it induces two trajectories: $t_1 = [sa, sb, sc]$ and $t_2 = [sd, se, sb, sc]$, with $sa : (x = 2, y = 2, qs = 1)$, $sb : (x = 3, y = 2, qs = 1)$, $sc : (x = 2, y = 2, qs = 2)$, $sd : (x = 2, y = 1, qs = 1)$, and $se : (x = 3, y = 1, qs = 1)$. Both trajectories are finite and end in a goal state. Informally, policy Π_2 consists in moving east at the beginning, and in setting qs to 2 as soon as a west move is performed from position $(3, 2)$.

Safety-oriented control problems In such problems, the objective is to find an applicable policy ensuring that the system is never blocked and that some properties are satisfied at each step. These properties are modeled by a *safety* relation R containing states s satisfying them. R may hold on unobservable attributes.

Definition 6 *A* safety-oriented *control problem is a pair* (M, R) *with* M *a control model and* $R \subset \mathbf{d}(S)$ *the safety relation. A solution to this control problem is an applicable policy* Π *for* M *such that all trajectories induced by* Π *are infinite and all states involved in these trajectories satisfy* R.

Let us consider a safety relation R imposing that position $(3, 1)$ is never reached (marked R on the figure). The safety-oriented control problem is (M, R), with M the control model defined previously and R the safety relation defined by constraint $\neg(x = 3 \wedge y = 1)$.

Memoryless policy Π_1 is a solution to this problem: it induces infinite trajectories which never reach position $(3, 1)$. Policy Π_2 is not a solution for two reasons: first it induces finite trajectories, and second one of these trajectories reaches position $(3, 1)$.

The main difference between goal-oriented and safety-oriented control problems is that, for the former, finite trajectories terminating in a goal state must be found whereas, for the latter, a control over an infinite horizon is sought. Goal-oriented control can be seen as a planning-like control, whereas safety-oriented control is rather related to model-checking or system specification.

Goal and safety-oriented control problems In such problems, the objective is to find an applicable policy so that all trajectories terminate in a goal state and satisfy some properties at each step.

Definition 7 *A* goal and safety-oriented *control problem is a triple* (M, G, R) *with* M *a control model,* $G \subset \mathbf{d}(S)$ *the goal relation, and* $R \subset \mathbf{d}(S)$ *the safety relation. A solution to this control problem is an applicable policy* Π *for* M *such that all trajectories* $[s_0, \ldots, s_n]$ *induced by* Π *are finite and verify* $G(s_n) = true$ *and* $\forall i \in [0..n], R(s_i) = true$.

Note that a goal and safety-oriented problem (M, G, R) is equivalent to the goal-oriented problem (M', G'), with M' resulting from the addition of R to F in M ($F' = F \wedge R$) and $G' = G \wedge R$ (safety requirements added to the feasibilities and to the goal).

4 DEPTH-FIRST SIMULATE AND BRANCH

4.1 General description

In order to solve control problems, we use a *Simulate and Branch* algorithm (SB). This algorithm performs the *depth-first* exploration of two trees:

- one tree T_B, called *policy branching tree*, which describes the possible decisions concerning the policy; each node n in T_B is labeled with an observation o and each branch coming out of n is labeled with a decision "$\Pi(o) = c$"; each node n in T_B can therefore be seen as a partial policy defined by the union of the branching decisions made on the path from the root to n;
- another tree T_S, called *system simulation tree*, which gives the possible evolutions of the controlled system given the current partial policy Π; children of the root node correspond to all possible initial states; each non-root node n in T_S is labeled with a state s; if $\mathbf{o}(s) \notin \mathbf{d}(\Pi)$, n is a leaf node; otherwise, the children of n are the possible successors of s induced by Π *i.e.*, states s' such that $T(s, \Pi(\mathbf{o}(s)), s')$ holds.

At each node in the policy branching tree, corresponding to a partial policy Π, an exploration of the system simulation tree is invoked. This exploration can return three different results:

1. a proof that policy Π is a solution to the control problem, when the traversal of the whole simulation tree reveals no inconsistency;
2. a proof of incompleteness of policy Π, when the exploration of T_S detects a reachable state s not covered by Π;
3. a proof of incorrectness of policy Π, when it is proved that no extension of Π can be a solution to the control problem.

In the first case, a solution policy is found. In the second case, a new branching node, associated with control $\Pi(\mathbf{o}(s))$, is introduced in the policy branching tree. In the third case, a backtrack occurs in the policy branching tree. The search terminates either when a solution policy is returned, or when the whole policy branching tree has been explored without finding a solution.

In order to save time in the exploration of T_S, we record, each time the exploration of T_S ends due to an uncovered state, the current search stack of the depth-first exploration of T_S. In order to save time in the exploration of T_B, we record in T_S the set D of decisions involved in the current trajectory. This allows a kind of *conflict-based backjumping* to be performed in T_B in case of inconsistency of the current trajectory (direct backtrack to the last decision involved in D). Storing search stacks and trajectory justifications can be memory consuming. If memory space is needed, both can be forgotten. The algorithm still works but may re-explore branches in T_S or backtrack in T_B on decisions not responsible for inconsistency.

Example Let us illustrate algorithm SB on the goal and safety-oriented control problem (M, G, R) with M the control model introduced in Section 2, G defined by $(x = 2 \wedge y = 2)$, and R by $\neg(x = 3 \wedge y = 1)$.

To answer this problem, algorithm SB uses the policy branching tree given in Figure 2 and the system simulation trees given in Figure 3. In Figure 3, the node at which a simulation tree exploration stops is represented by a plain line box. The nodes associated with solved states (starting from these states, the current policy is a solution) are represented by a dotted box.

Search begins at the root of the policy tree. At this point, the current policy is empty. The first step is to explore the possible evolutions of the system. This corresponds to simulation tree $SimuA$. The exploration of $SimuA$ stops in initial state $(x = 2, y = 2, w_N, w_S, qs = 1)$ which is not covered by the current empty policy. A decision to do nothing in this state $(\Pi(w_N, w_S, qs = 1) = \epsilon)$ is made in the policy tree. Exploration of $SimuB$ then reveals an inconsistency since, in state $s : (x = 2, y = 1, w_N, w_S, qs = 1)$, the current policy indicates to do nothing, but s is not a goal state. Backtrack occurs in the policy tree and another decision $(\Pi(w_N, w_S, qs = 1) = (m = m_E, qc = 1))$ is tried. Exploration of $SimuC$ then shows that state $(x = 3, y = 2, w_N, w_E, qs = 1)$ is uncovered. A decision is made in the policy tree. Exploration of $SimuD$ reveals an inconsistency (loopy trajectory). Backtrack occurs in the policy branching tree and another choice is made. Exploration of $SimuE$ detects an uncovered state, for which a decision is made. An inconsistency is then revealed in $SimuF$ since the current policy can lead to the undesirable position $(3, 1)$. Because this state is reached in $SimuF$ using only decision $\Pi(w_N, w_S, qs = 1)$, a backjump can occur in the policy tree directly to this decision. The search continues until a solution policy is found.

In practice, each time a new uncovered state is discovered, the pending nodes in the depth-first exploration of the current simulation tree are stored. For instance, $SimuF$ is obtained starting from $SimuE$. It must also be stressed that, due to partial observability, an inconsistency revealed in a branch of a simulation tree can be induced by a decision made in a completely different branch. See for instance the inconsistency revealed in $SimuF$ which is due to the decision made for the uncovered state in $SimuA$.

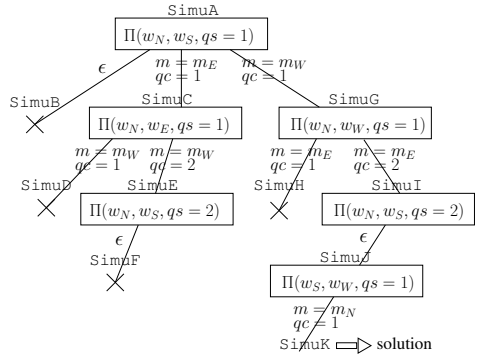

Figure 2. Policy branching tree

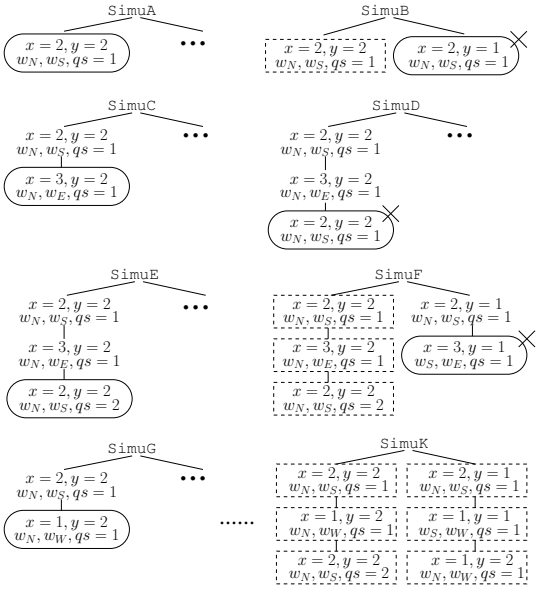

Figure 3. System simulation trees

4.2 Pseudo-code

Due to space limitations, proofs are omitted, and the pseudo-code is presented in the context of goal-oriented control problems and without backjumping. The main function in Algorithm 1 is **SB**. It takes as inputs a policy Π, a function M associating with each state s a mark $M(s) \in \{\mathit{SOLVED}, \mathit{PROCESSING}, \mathit{NONE}\}$, and a search stack L. If $M(s) = \mathit{SOLVED}$, the current policy is a solution starting from s. If $M(s) = \mathit{NONE}$, s has not been considered yet. Otherwise, $M(s) = \mathit{PROCESSING}$. A search stack L is an ordered list of search nodes which are pairs (s, U), where s is a state and U the set of immediate successor states of s not explored yet (pending nodes).

Function $\mathbf{SB}$ first calls function $\mathbf{simulate}$ to explore the simulation tree, using current policy Π. If this exploration proves that Π is a solution, then $\mathbf{SB}$ returns $true$ and Π. If Π is not a solution and no uncovered state has been revealed, it returns $false$. Otherwise, a new branching choice is introduced for observation $\mathbf{o}(s)$ associated with the uncovered state s, and $\mathbf{SB}$ is recursively called. When s satisfies the goal, a possible branching choice is $\Pi(\mathbf{o}(s)) = \epsilon$. Such a choice forces $\Pi(\mathbf{o}(s))$ to be undefined. By convention, we consider that $T(s, \epsilon, s')$ is false for every s'.

Function $\mathbf{simulate}$ takes as input the same parameters as function $\mathbf{SB}$ does. It continues the depth-first simulation tree exploration by calling function $\mathbf{simuAllSucc}$ for each search node (s, U) in the search stack. Function $\mathbf{simuAllSucc}$ calls function $\mathbf{simuOneSucc}$ to successively explore each state in U. When all successors of s have been considered, s is marked $SOLVED$.

Function $\mathbf{simuOneSucc}$ takes as input current policy Π, current marks M, and a single state s. If s is already solved, then the exploration succeeds. If s is already processing, then a loop inducing an infinite trajectory is detected, hence inconsistency is returned. If $\mathbf{o}(s)$ is not covered by Π, then $\mathbf{simuOneSucc}$ returns that s is uncovered. When $\mathbf{o}(s)$ is covered by Π, but the associated decision is not feasible, an inconsistency is returned, as well as when the decision is ϵ, but s is not a goal state. In other cases, s is marked $PROCESSING$ and the exploration of successors of s is triggered.

The initial call is $\mathbf{SB}(\Pi_0, M_0, L_0)$ with Π_0 the empty policy, M_0 the function associating mark $NONE$ with every state, and L_0 the search stack reduced to search node $(\cdot, \{s \in \mathbf{d}(S) \mid I(s)\})$ representing all possible initial states.

Proposition 1 $\mathbf{SB}$ *is sound and complete: if the goal-oriented control problem has a solution, then initial call* $\mathbf{SB}(\Pi_0, M_0, L_0)$ *returns* $(true, \Pi)$ *with* Π *a solution policy; otherwise, it returns* $(false, \Pi_0)$.

Proposition 2 *Computing* $\mathbf{SB}(\Pi_0, M_0, L_0)$ *is time* $O(|\mathbf{d}(S)|^2 \cdot |\mathbf{d}(O)|^{|\mathbf{d}(C)|+1})$. *When search stacks are forgotten, computing* $\mathbf{SB}(\Pi_0, M_0, L_0)$ *is space* $O(|\mathbf{d}(O)| \cdot |\mathbf{d}(C)| + |\mathbf{d}(S)|^2)$.

Algorithm $\mathbf{SB}$ is a pure search exploration procedure which uses no classical planning heuristics. We however use three simple settings in the policy branching tree when choosing a value for $\Pi(\mathbf{o}(s))$:

- when $G(s)$ holds, control ϵ (undefined policy) is tried first, in order to build policies which halt trajectories as soon as possible;
- controls having qc as small as possible are tried first, in order to obtain controllers as memoryless as possible;
- if $qcmax$ denotes the maximum value used for qc in Π, then branching choices where $qc > qcmax + 1$ are forbidden; this allows to avoid exploring symmetric solutions by forcing the controller to consider new internal states in an ascending order.

Extension to Other Control Problems Algorithm $\mathbf{SB}$ is generic and reusable for other control problems. When the goal holds only on observable attributes, a faster algorithm can be obtained by removing option ϵ from the branching choices and by replacing in function $\mathbf{simuOneSucc}$ tests 34 and 40 by:

34' $(M(s) = SOLVED) \vee G(s)$

40' $\neg F(s, \Pi(\mathbf{o}(s)))$

For safety-oriented control problems, loops inducing infinite trajectories are allowed and it suffices to replace in function $\mathbf{simuOneSucc}$ tests 34, 36, and 40 by:

34" $(M(s) = SOLVED) \vee (M(s) = PROCESSING)$

36" $\neg R(s)$

40" $\neg F(s, \Pi(\mathbf{o}(s)))$

Algorithm 1: SB algorithm for goal-oriented control problems

```
1  SB(Π, M, L) begin
2      (M', L') ← (M, L)
3      (ok, s, M', L') ← simulate(Π, M', L')
4      if ok then return (true, Π)
5      else if s = null then return (false, Π)
6      else
7          M'(s) ← PROCESSING
8          Dc ← {c ∈ d(C) | F(s, c)}
9          if G(s) then Dc ← Dc ∪ {ε}
10         while Dc ≠ ∅ do
11             choose c ∈ Dc; Dc ← Dc \ {c}
12             Π' ← Π ∪ {o(s) → c}
13             Ds ← {s' ∈ d(S) | T(s, c, s')}
14             (ok, Π') ← SB(Π', M', (s, Ds).L')
15             if ok then return (true, Π')
16         return (false, Π)
17 end

18 simulate(Π, M, L) begin
19     while L ≠ ∅ do
20         (s, U) ← first(L); delete_first(L)
21         (ok, uncov, M, L') ← simuAllSucc(Π, M, s, U)
22         if ¬ok then return (false, uncov, M, L'.L)
23     return (true, null, M, ∅)
24 end

25 simuAllSucc(Π, M, s, U) begin
26     while U ≠ ∅ do
27         choose s' ∈ U; U ← U − {s'}
28         (ok, uncov, M, L') = simuOneSucc(Π, M, s')
29         if ¬ok then return (false, uncov, M, L'.(s, U))
30     M(s) ← SOLVED
31     return (true, null, M, ∅)
32 end

33 simuOneSucc(Π, M, s) begin
34     if M(s) = SOLVED then
35         return (true, null, M, ∅)
36     else if M(s) = PROCESSING then
37         return (false, null, M, ∅)
38     else if o(s) ∉ D_Π then
39         return (false, s, M, ∅)
40     else if ¬F(s, Π(o(s))) ∨ (Π(o(s)) = ε ∧ ¬G(s)) then
41         return (false, null, M, ∅)
42     else
43         M(s) ← PROCESSING
44         Ds ← {s' ∈ d(S) | T(s, Π(o(s)), s')}
45         return simuAllSucc(Π, M, s, Ds)
46 end
```

5 EXPERIMENTS

We ran our experiments on a Xeon processor 2GHz, 1GB RAM. The solver implementing the SB procedure is called *Dyncode* and uses the Gecode constraint programming library [5]. Models used were directly written in a constraint-based form (they were not parsed from PDDL). The different relations were expressed and handled as sets of constraints and not as large tables. We used hash tables to record decision policy Π, states marks M, and sets of unexplored states U for each element (s, U) in a search stack L. The data structures used are backtrackable in the sense that we record modifications made on Π, M, and L, instead of copying these structures over and over.

We first compared Dyncode with the planner defined in [3] for synthesizing finite-state controllers, which we will refer to as BPG. This planner solves contingent planning problems by translating them into classical planning and by using classical planners. We did not rerun BPG, which is not publicly available, but simply took the CPU times given in [3], obtained on a Xeon 1.86GHz, 2GB RAM. For the comparison to be fair, we provided Dyncode with the minimum value of N for each instance and searched for plans reaching the goal but not necessarily halting in a goal state, as done by BPG. In the constraint-based models used by Dyncode, a few preconditions easily expressible in PDDL are added, e.g. to forbid moves towards walls. BPG cannot directly handle such preconditions. Table 1 shows that Dyncode always runs as fast or faster than BPG (the "as fast" statement integrates the fact that CPU times for BPG have only one significant digit). Some instances solved in tens of minutes by BPG are solved by Dyncode in less than one second or in a few seconds. Several explanations can be provided. First, with Dyncode, the few preconditions added allow the search space to be pruned earlier. Second, the translation approach used by BPG may hidden some features of the problem to be solved, contrarily to the Simulate and Branch procedure used by Dyncode which is directly suited for control problems. Third, generic heuristics described at the end of Section 4 help Dyncode handling the controller memory part.

Problem	Instance	N	CPU time (sec.)		
			BPG	Dyncode	Proof $< N$
Hall-A	1×4	2	0.0	0.01	0.02
	4×4	4	5730.5	0.26	2.35
Hall-R	1×4	1	0.0	0.01	0
	4×4	1	0.0	0.02	0
Prize-A	4×4	1	0.0	0.02	0
Corner-A	4×4	1	0.1	0.02	0
Prize-R	3×3	2	0.1	0.03	0.03
	5×5	3	2.7	2.37	0.97
Corner-R	2×2	1	0.0	0.01	0
	5×5	1	1.6	0.02	0
Prize-T	3×3	1	0.1	0.05	0
	5×5	1	0.3	0.34	0
Blocks	6	2	0.8	0.02	0.02
	20	2	34.8	0.04	0.02
Visual-M	$(8, 5)$	2	1289.5	3.59	0.27
Gripper	$(3, 5)$	2	4996.1	0.06	0.02

Table 1. Comparison Dyncode vs. BPG; N: min memory of a solution policy; Proof$< N$: Dyncode proof that no solution with memory $< N$ exists

We then compared Dyncode with MBP [1], a planner able to handle non determinism and partial observability. Basically, MBP performs a forward search in the space of belief states. It branches on decisions when a new belief state is reached, and records marks on belief states (instead of marks on states). MBP also uses BDDs to limit state space explosion. It produces full-recording controllers represented as conditional plans. Experiments were performed on all domains presented in [1]. The constraint-based models used by Dyncode contained the same knowledge as those used by MBP. Figure 4 gives the results obtained for domains Emptyroom, Maze, and Ring. The minimum value of N is provided to Dyncode.[2] Figure 4 shows that in terms of CPU time, MBP clearly outperforms Dyncode. This is due to the fact that MBP uses BDDs techniques and searches in the space of belief states, whereas Dyncode explores all reachable states individually. But in terms of memory size of the controllers

[2] Emptyroom: $N = 0$; Maze: $N = 2$ for instances 3 to 7, $N = 3$ for instances 9 to 13; Ring: $N = k$ for instance k.

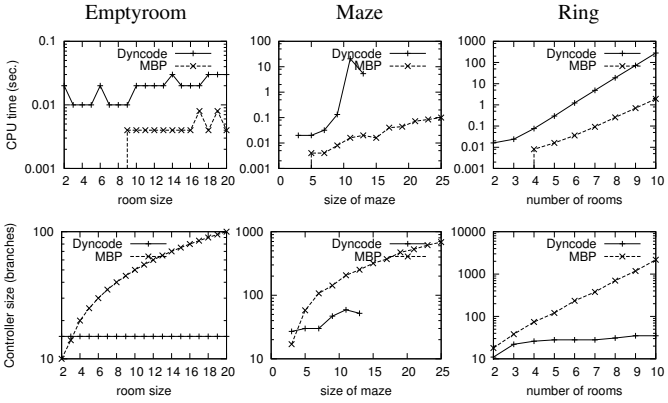

Figure 4. Comparison Dyncode vs. MBP; first line: CPU time to synthesize a solution controller (1h timeout); second line: controller size, expressed for MBP as the number of arcs in the conditional plan it produces and for Dyncode as the size of a BDD representing the controller produced

produced, Dyncode clearly outperforms MBP (for domain Emptyroom, the size of the controllers produced does not even depend on the instance size). This is due to the fact that MBP computes full-recording controllers which may record useless features (reasoning on belief states is sufficient *but not necessary* to act in a non deterministic domain). As a result, with our algorithmic approach, the price to pay for obtaining compact controllers is CPU time. For embedded controllers, this price must however be paid only offline.

6 CONCLUSION

This paper presented an approach for synthesizing finite-state controllers in non-deterministic and partially observable domains. This approach uses a depth-first simulate and branch algorithm applicable to several control problems, from goal-oriented control to safety-oriented control. Compared to existing work, several hypotheses were relaxed and significant gains were obtained in terms of either CPU time, or memory size of the controllers produced. Future works will concern ways to speed search, e.g. using planning heuristics or BDDs, and the extension towards optimization or stochastic aspects.

REFERENCES

[1] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, 'Planning in Non-deterministic Domains under Partial Observability via Symbolic Model Checking', in *Proc. of IJCAI-01*.

[2] B. Bonet and H. Geffner, 'Planning with Incomplete Information as Heuristic Search in Belief Space', in *Proc. of AIPS-00*.

[3] B. Bonet, H. Palacios, and H. Geffner, 'Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners', in *Proc. of ICAPS-09*.

[4] E. Emerson, 'Temporal and Modal Logic', in *Handbook of Theoretical Computer Science, Volume B:Formal Models and Semantics*, 995–1072, Elsevier, (1990).

[5] Gecode Team. Gecode: Generic Constraint Development Environment, 2006. Available from `http://www.gecode.org`.

[6] L. Kaelbling, M. Littman, and A. Cassandra, 'Planning and Acting in Partially Observable Stochastic Domains', *Artificial Intelligence*, **101**, 99–134, (1998).

[7] N. Meuleau, L. Peshkin, and L. Kaelbling, 'Learning Finite-State Controllers for Partially Observable Environments', in *Proc. of UAI-99*.

[8] P. Poupart and C. Boutilier, 'Bounded Finite State Controllers', in *Proc. of NIPS-03*.

[9] M. Puterman, *Markov Decision Processes, Discrete Stochastic Dynamic Programming*, John Wiley & Sons, 1994.